高阶函数处理可以将函数作为参数传递外，还可以把函数作为结果值返回：

闭包：

在一个内部函数中引用外部作用域的变量，但这个变量不在全局作用域里，则这个内部函数就是一个闭包。外部函数返回内部函数时，相关的变量和参数都保存在返回的内部函数中，这种称为闭包。

闭包的优点或者用处：

1、在函数外部可以调用函时内部的变量

2、让这些变量的值始终保留在内存中（当前运行环境）

```
def lazy_sum(*args):
    def sum():
        ax = 0
        for i in args:
            ax = i + ax
        return ax
    return sum
```

调用lazy_sum()返回的是**求和函数**，而不是求和结果。

闭包的常见误区：

1、尝试在闭包中改变外部作用域的局部变量

```
def foo():
    a = 1
    def bar():
        # 右侧的a为外部变量，左侧的a为内部变量
        a = a + 1
        return a
    return bar()

>>> c = foo()
>>> print c()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 4, in bar
UnboundLocalError: local variable 'a' referenced before assignment
>>> def wrapper():
...     a = 1
...     def inner():
...             return a
...     return inner
...
```

```
>>> f = wrapper()
>>> f()
1
>>> def wrapper():
...     a =1
...     def inner():
...             a= a+1
...             return a
...     return inner()
...
>>> f1 = wrapper()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 6, in wrapper
  File "<stdin>", line 4, in inner
  # 局部变量与外部变量重名造成的错误
UnboundLocalError: local variable 'a' referenced before assignment



>>> def wrapper():
...     a =1
...     def inner():
...             nonlocal a
...             a = a+1
...             return a
...     return inner()
...
>>> f2 = wrapper()
>>> f2()
>>> print(f2)
2
>>> a = x +1

>>> def wrapper():
...     a =1
...     def inner():
...             b=2
...             a = a
...             a = a+2
...             return a
...     return inner
...
>>> f4 = wrapper()
>>> f4()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
  File "<stdin>", line 5, in inner
UnboundLocalError: local variable 'a' referenced before assignment
>>>
```

解决办法有两个：

方法一、将a设置为一个容器，比如表list;

方法二、将a声明为nonlocal变量（python3支持），nonlocal会从从上一层的环境中寻找这个变量、

```
def foo():
    a = 1
    b = [1]
    def bar():
        nonlocal a
        a = a + 1
        b[0] = b[0] + 1
        return a, b[0]
    return bar()
```

2、误以为返回的内部函数已经执行，对执行结果误判。

```
def count():
    fs = []
    for i in range(1, 4):
        def f():
            return i * i
        fs.append(f)
    return fs
f1, f2, f3 = count()


>>> f1()
9
>>> f2()
9
>>> f3()
9
```

全部都是9！原因就在于返回的函数引用了变量i，**但它并非立刻执行**。等到3个函数都返回时，它们所引用的变量i已经变成了3，因此最终结果为9。

返回闭包时牢记一点：返回函数不要引用任何循环变量，或者后续会发生变化的变量。

如果一定要引用循环变量，方法是再创建一个函数，用该函数的参数绑定循环变量当前的值，无论该循环变量后续如何更改，已绑定到函数参数的值不变：

```
def f(j):
    def g():
        return j * j
    return g
    fs = []
    for i in range(1, 4):
        fs.append(f(i))#f(i)被立刻执行，因此i的当前值被传入f()
    return fs

>>> f1, f2, f3 = count()
>>> f1()
1
>>> f2()
4
>>> f3()
9
```